SYSTEMS ENGINEERING · WEEK 8 · STATION III

Concurrency

Threads, locks, load-reserved and store-conditional. Interleavings are a state space, and it explodes.

THREADS

A thread is a process that shares the address space. Only the stack and the registers are its own.

Fork copies pages. A thread shares them: same page table, same code, same data, same heap, a fresh stack. In selfie a thread is a context whose page table points to the parent's, with its own registers and a stack segment of its own. That is the whole difference, and it is a difference in how much isolation you keep.

Less isolation, cheaper communication: a global variable is a message. And the timer of week 5, invisible to a process, is now visible: it can fire between any two instructions, and the other thread sees memory as it was at that instant.

```
// x = x + 1, as the compiler emits it
ld t0,-16(gp) // read x
addi t0,t0,1 // ← the timer can fire here
sd t0,-16(gp) // write x
// two threads, two increments, x goes up by one
```

THE STATE SPACE

Two threads of *n* instructions have C(2n, n) interleavings. The meaning of the program is the set of all of them.

20

INTERLEAVINGS OF TWO 3-INSTRUCTION THREADS

≈ 10^{29}

OF TWO 50-INSTRUCTION THREADS

2^{35}

BITS OF STATE PER THREAD, BEFORE INTERLEAVING

The Meaning chapter defined a program's meaning as the state it computes. With threads, the program has one meaning per interleaving, and the scheduler picks which. A program is correct if every interleaving is, and there are more of them than tests. The state-space explosion of the Size chapter, arriving from a new direction.

So concurrency is the second place in this class where testing shows presence and not absence, and where the honest tools are a proof for all interleavings, or a bound. Locks are the way to make most of the interleavings equivalent, so that fewer need to be considered.

LOCKS

Mutual exclusion: at most one thread in the critical section. Acquiring the lock must itself be atomic.

A lock is a word: 0 free, 1 taken. Acquire: wait until it is 0, then set it to 1. Release: set it to 0. Between acquire and release the thread is alone with the shared data, and the interleavings inside the section collapse to one.

But "wait until 0, then set 1" is a load, a branch and a store: three instructions, and the timer can fire between them. Two threads both see 0 and both take the lock. The lock has the race it was meant to remove. No sequence of ordinary instructions fixes this; the machine has to offer one that cannot be interleaved.

BLOCKING

A thread that finds the lock taken can spin, burning its slice, or ask the kernel to block it: the third process state of week 5, now entered on a lock. The kernel wakes it on release. And two threads each holding a lock the other waits for are both blocked, forever: deadlock, which the kernel cannot see without looking for cycles, and which week 12 places on the axis.

ATOMIC INSTRUCTIONS

`lr.d` reserves. `sc.d` succeeds only if nobody wrote in between.

```
// x = x + 1, lock-free
retry:
lr.d t0,(a0) // load x, reserve a0
addi t0,t0,1
sc.d t1,t0,(a0) // store if still reserved; t1 = 0 on success
bne t1,zero,retry // someone interfered: try again
```

Load-reserved and store-conditional, RISC-V's pair: the load remembers the address, the store checks that no other core or thread wrote to it since, and fails instead of overwriting. A failed store is not an error; it is information, and the loop uses it.

RISC-U does not have them. Your assignment adds them to the machine, the emulator, and the hypervisor: a new instruction is a new line in the machine chapter's table, a new case in mipster's `execute`, and a new column in the interleaving state space that behaves.

With these, a lock is four instructions and correct. Without them, no lock is. The whole of concurrent programming rests on one instruction that the machine promises not to interleave.

PROGRESS

Lock-free is a progress guarantee, not the absence of locks.

BLOCKING

A thread holding a lock that is preempted, or dies, stops every thread waiting for it. Progress depends on the scheduler being kind.

LOCK-FREE

Some thread always completes in a bounded number of steps of the system. The lr/sc loop: if my store fails, someone else's succeeded. Livelock is possible; deadlock is not.

WAIT-FREE

Every thread completes in a bounded number of its own steps. Stronger, rarer, and the only one that is a bound on a single thread rather than on the system. Bounds again, with different quantifiers.

Progress properties are the concurrency chapter's version of liveness: statements about the future of an execution, hence about all interleavings, hence exactly what testing cannot show.

ASSIGNMENTS

lock and threads.

1. `lock`: system calls `lock()` and `unlock()` on one global lock in the kernel. Acquire blocks the calling context; release wakes one waiter. The kernel is single-threaded, so no atomic instruction is needed here: the door is the atomicity.
2. `threads`: `pthread_create`, `pthread_join`, `pthread_exit`: a context that shares the caller's page table and gets its own stack. Join blocks like wait. Decide what happens to a process's threads when it exits.
3. Run two threads incrementing one global without a lock and watch the count; then with your lock. Then make the timeout one instruction and watch again.
4. `./grader/self.py lock` and `./grader/self.py threads`.

NEXT WEEK

Atomic instructions in the machine: `lr.d` and `sc.d` in RISC-U, mipster and hypster, and a thread-safe `malloc` built on them. Keep your threads; they are the test harness.